Persistent Stores and Hybrid systems*

R. L. Grossman† D. Valsamis, and X. Qin

Laboratory for Advanced Computing (m/c 249)
University of Illinois at Chicago
851 South Morgan Street
Chicago, IL 60607

Abstract

We describe a proof of concept implementation of software tools for the simulation, analysis, and real-time control of flows of hybrid systems. Using these tools, one can create persistent object stores containing trajectories of hybrid systems and control such systems with an appropriate query on the store. To illustrate these ideas, we describe how to use object stores to solve path planning problems for hybrid systems.

1. Introduction

This paper describes a proof of concept implementation of software tools for the simulation, analysis, and real-time control of flows of hybrid systems. Using these tools, one can create persistent object stores containing trajectories of hybrid systems and control such systems with an appropriate query on the store. To illustrate these ideas, we describe how to use object stores to solve path planning problems for hybrid systems.

By a hybrid system, we mean a collection of nonlinear control systems, each corresponding to a mode of the hybrid system, with mode switching determined by a finite state automaton, reacting to discrete input events. By a persistent object store [2], we mean a collection of complex objects which are persistent. Objects are called persistent if they can be accessed and queried independently of the processes which create them.

For the simplest applications, the persistent object store is populated with trajectory segments from each of the nonlinear systems corresponding to the different modes of the hybrid system. Flows of the hybrid system can be computed a "system at a time" by querying the object store as follows:

1. The query retrieves the desired trajectory segment belonging to the nonlinear system corresponding to the initial mode.

2. An input event is accepted and a new mode computed by stepping the automaton or executing the appropriate software.

3. The query proceeds by selecting the desired trajectory segment belonging to the nonlinear system corresponding to the new mode. Following this, a new input event is accepted and the cycle repeats.

For other applications, it is appropriate to populate the store with hybrid trajectory segments themselves. Longer hybrid trajectory segments, corresponding to longer input event sequences, can then be obtained by the appropriate query on the store.

With the appropriate design of the store, a trajectory segment which is expensive to compute can be retrieved at the same cost as a trajectory segment which is inexpensive to compute. For this reason, viewing control algorithms as appropriate queries on object stores of trajectory segments is competitive for problems in which it is expensive to compute the appropriate trajectory, such as path planning problems, and for problems in which nearby trajectory segments are also desired, such as in some approaches to dealing with model uncertainty.

Our viewpoint for developing software tools for hybrid systems is closely related to the viewpoint of Back, Guckenheimer and Nerode who have extended dstool in order to analyze the phase portraits of hybrid systems [10]. Our viewpoint for modeling hybrid systems is closely related to the viewpoint of Meyer [12] and of Kohn and Nerode [11]. In this latter work continous control systems are also coupled to discrete automata, but the automata are used to extract control laws for the continous systems rather than to switch modes between them.

*This research was supported in part by NASA grant NAG2-513 and DOE grant DE-FG02-92ER25133.

†Please send correspondence to Robert Grossman, grossman@math.uic.edu

2. Path planning

In this section, we illustrate this approach by describing how path planning algorithms can be viewed as queries on an object store of trajectory segments, following [7]. There are three distinct phases:

1. In a precomputation, the object store is populated with short duration trajectory segments, each representing a reference trajectory to be followed using a regulator.

2. In a precomputation, each trajectory segment is assigned a sequence of indices.

3. The input to each query is the desired flight path and a tolerance. The output of a successful query is a sequence of trajectory segments with the property that they approximate the desired flight path to within the tolerance. The controls necessary to generate each trajectory segment are attached to the trajectory segment and can be used as the basis for a robust control algorithm to follow the retrieved reference trajectory.

The retrieved trajectory segments are computed as follows:

Break up. The query path is broken up into a number of smaller query path segments which are placed on a stack and the index of the query path segment is computed.

Retrieve. For each query path segment on the stack, the query path is removed from the stack and all trajectory segments in the store with the same index are retrieved and compared to the query path segment. If one of them matches the query path segment to within the desired tolerance, the trajectory segment is returned with the query. If not, the query path segment is broken up again and the algorithm continues recursively.

Each trajectory segment and query path segment has a sequence of indices attached to it. If a query path segment has been broken up n times by the algorithm, the nth index is used. It may happen that the algorithm does not terminate successfully. In this case, it returns the best approximating sequence of trajectory segments in the store. The threshold can then be adjusted or additional trajectory segments added to the store to provide better approximating sequences of trajectory segments.

This basic algorithm is easy to adapt to hybrid systems: one simply adds a mode attribute to each trajectory segment, steps an automaton with each trajectory segment retrieved, and only retrieve trajectory segments of the next required mode by using a suitable index function.

Path planning algorithms for control systems generally work by concatenating trajectory segments of a fixed, specialized type. Restricting the types of trajectories used to a small enough class provides enough structure so that controls can be computed to approximate the path. It is usually quite expensive to compute these controls. For example, a path planning algorithm by Murray and Sastry [13] employs trajectories which are sinusoids at integrally related frequencies. On the other hand, using the software tool described here, it is easy to make use of large numbers of precomputed trajectory segments of a general type. Computing the path requires only a low cost selection of the most appropriate trajectory segments. With the proper index function, any trajectory segment can be retrieved with constant cost, independent of the number of trajectory segments. By using very general classes of trajectory segments, it becomes easier to match the desired path. In essence, space is traded for time and precomputation is traded for computation: large amounts of space are required to store all the precomputed trajectory segments and large amounts of time are required to populate the store and compute the required indices, but the cost to approximate the path is low.

3. Persistent Stores

An object manager creates, stores, and accesses persistent stores of complex objects [2], such as the TrajectorySegment object described in Figure 1. For example, a TrajectorySegment is a complex object consisting of several subobjects, including a list of points along the trajectory, a list of parameter and control values describing the control system which produced the trajectory segment, an integer defining the mode of the trajectory segment, and a list of indices used to retrieve the trajectory segment.

From one viewpoint, an object manager is the analogy of a file manager, while a persistent store is the analogy of a file system. There are several reasons to use an object manager to manage scientific data rather than a file manager:

— Scientific data often has a complex structure which easily fits into an object data model, while it must be "flattened" to fit into a file.

— By attaching methods to the objects, it is easy to provide application specific indexing and access methods in order to provide higher performance access to the data.

— Algorithms to manage and analyze scientific data are often complex. There is usually a "low-impedance" between the programming language used to implement the algorithms and the language used to create, store, and access the objects.

Our initial prototype of the system described below was implemented using a commercial relational database: we now use an internally developed software tool providing persistence, called ptool, for these, and related, reasons.

Finally, we mention several requirements for an object manager to be useful in these types of applications:

— The object manager must provide low overhead and high performance access to objects. This is in contrast to object managers for other types of applications which must provide "safe" access to objects, for example by using a transaction model.

— The object manager must scale as the number of objects grows, as their size grows, and as the complexity of the query grows. Again, object managers for non-scientific applications usually do not require the ability to work with large numbers of objects, nor are their queries usually as numerically intensive.

The object manager we designed and implemented satisfies these reqirements.

4. Hybrid Systems

In the sections above, we have explained how to create and access persistent object stores of trajectory segments, or flows, from hybrid systems. In this section, we explain a little of the theoretical background for analyzing flows of hybrid systems.

We explain how to describe hybrid systems from the state space and observation space viewpoints, following [6] and [5].

Let k denote a field of characteristic 0, say the real numbers. To define a hybrid system in the state space representation, we need to specify

1. A collection of control systems. Consider a collection of control systems evolving on the common state space X of the form

$$\dot{x}(t) = u_1(t)E_1^{(i)}(x(t)) + u_2(t)E_2^{(i)}(x(t)),$$

```
class TrajectorySegment {
   int dimension;
   int length_of_segment;
   int number_of_controls;
   int number_of_parameters;
   int number_of_indices;
   int mode;
   int* index;
   float* parameters;
   float* controls;
   float* points;
};
```

Figure 1: A TrajectorySegment object consists of sequence of points, a sequence of controls and parameters, a mode, and some additional information.

$$x(0) = y^{(i)} \in X, \quad i = 1, \ldots, n,$$

where $E_1^{(i)}$ and $E_2^{(i)}$ denote vector fields describing the dynamics of control system i and $t \to u_j(t)$ are the controls.

2. A finite state automaton. Consider an automaton on the states $s_1, \ldots, s_m$. The automaton accepts input symbols α from a finite set of input symbols Ω, each corresponding to a discrete event, and changes its state from s_i to $s_j = s_i \cdot \alpha$. Let Ω^* denote the semigroup of words generated by the input symbols $\alpha \in \Omega$.

3. A mode interpretation. We assume that each state s_j of the automaton is associated with one of the control systems.

Flows of the hybrid system in this representation are simply formal concatenations of flows from the various control systems and the automaton.

We turn now to the observation space representation, which turns out to be convenient for studying various properties of flows [5]. The basic idea is to take as fundamental the space of observations of the system rather than the space of states. The action of the dynamics on the state space translates into an action of the dynamics on the observation space. Broadly speaking, the observation space representation is dual to the state space representation. This viewpoint has been used to describe discrete time systems by Sontag [14] and continuous time systems by Bartosiewicz [1]. This is also closely connected to the emphasis on observations rather than states central to quantum mechanics.

Given the data above, we define the observation space representation to be the pair consisting of the algebra of observation functions

$$R = \{f : X \longrightarrow k\},$$

and an algebra coding the dynamics

$$H = k<\xi_1, \xi_2> \amalg k\Omega^*$$

together with an action of H on R. Here $\amalg$ denotes the free product of the algebras, $k\Omega^*$ is the semigroup algebra, and $k<\xi_1, \xi_2>$ is the free associative algebra. For concreteness, let the state space X be $\mathbf{R}^N$, let the observation space be the ring of polynomials $k[X_1, \ldots, X_N]$ and assume that the dynamics are defined by vector fields with polynomial coefficients. To define the action of H on R, consider a typical element of H

$$\xi_2\alpha_3(\xi_1 + 2\xi_2)\alpha_2.$$

Reading from right to left, the input symbol α_2 causes the automaton to change its state; associated with the new state is a control system, say number 3; we interpret $\xi_1 + 2\xi_2$ as the differential operator $E_1^{(3)} + 2E_2^{(3)}$, which acts upon a polynomial $f(X_1, \ldots, X_N)$ as usual to give a new polynomial g; the next input symbol α_3 causes the automaton to change its state again, say the control system associated with this new state is number 1; the corresponding action on g is then $E_2^{(1)}$.

For the formal definition of the action, see [6]. In the applications described in the sections above, we are interested in elements of $k<\xi_1, \xi_2>$ of the form

$$\exp(u_1(t)\xi_1 + u_2(t)\xi_2)$$

corresponding to flows of the appropriate control system. These elements have nice algebraic properties (they are group-like [5]), which can be used to study their properties.

5. Implementation and Experiments

To test these ideas, we developed a persistent object manager called ptool [8], [4] and [9] which we used to populate a persistent object store of trajectory segments. Ptool was designed to provide low overhead, high performance access to large numbers of objects in a distributed high performance computing environment and satisfies the requirements described in the section on persistent stores above.

We also developed companion tools to populate stores of trajectory segments and retrieve trajectoy segments which approximate a given path.

select.	0.5%		1%		3%	
system	user	cpu	user	cpu	user	cpu
car	13.75	9.25	25.25	4.75	79	66
robot	2.25	2.75	4.5	3.75	7	10

Table 1: This table describes experiments on two different trajectory stores. Both systems are taken from [13]. The first system is a four dimensional model of a kinematic car. The store consists of 396,000 trajectory segments and is 115.6 MBytes in size. The second system is a three dimensional model of a hopping robot. The store consists of 121,500 trajectory segments and is 31.1 MBytes in size. The table contains the time in ticks to retrieve the indicated percentages of the trajectory segments for secondary testing. A tick is 1/60 seconds. The times for queries retrieving 0.5% and 1.0% of the trajectory segments are averages of four different queries each.

For the experiments here, we used a Sun Sparcstation 1. Experiments were done on model systems in dimensions three through six taken from [13] and [12]. The stores ranged in size from approximately 10 MBytes to 200 MBytes and contained between 100,000 and 1,000,000 trajectory segments. The trajectory segments were obtained by using a 5th order Runge-Kutta algorithm. The path planning algorithm used three levels of indices.

Table 1 summarizes some of the experiments performed. With the indices we used, we expected to be able to retrieve any given trajectory in constant time, say k, and to retrieve any given set of n trajectories in time kn. To test this, we create several stores and retrieved sets of trajectories containing 0.5%, 1%, 3%, 6%, 12%, 25%, 50%, and 100% of the trajectories. For example, for the kinematic car, ptool retrieved 0.5% of the 396,000 stored persistent trajectory segments in 23 ticks, where a tick is 1/60 seconds. This seems to provide sufficient performance for real time control of hybrid systems, a topic we are currently exploring, and scales as expected.

We choose to experiment with these relatively small stores because of system limitations: from other experiments, we expect the numbers to scale linearly for stores that are up to one or two magnitudes larger.

6. Conclusions

The work described here demonstrates the feasibility of using persistent stores of trajectory segments to simulate, analyze, and control hybrid systems. Current work is focusing on implementing more complex examples, using data centered parallelism to obtain